پایگاه داده‌ها

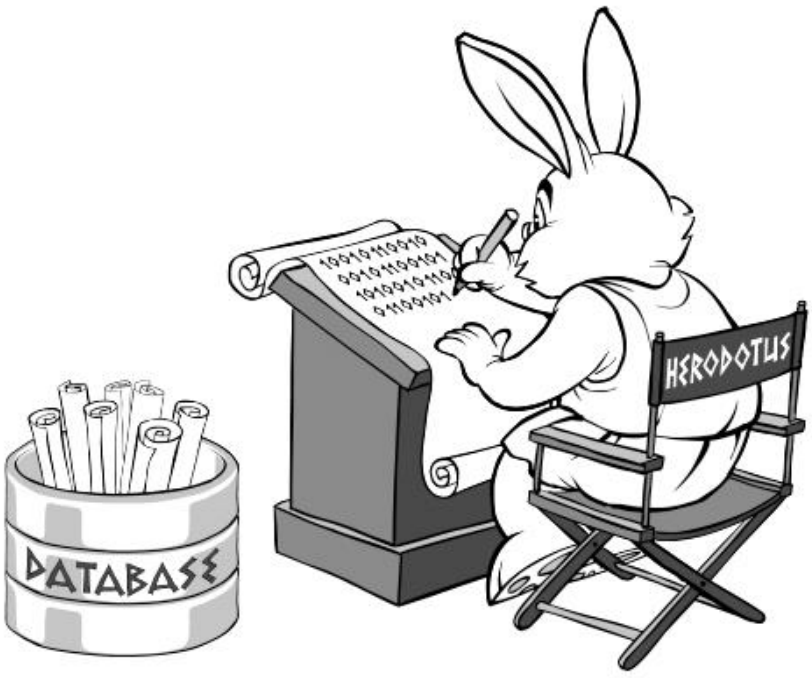
مدرس: میلاد وزان

دانشگاه شهید بهشتی - دانشکده ریاضی - گروه آموزشی آمار

نیم‌سال دوم ۱۴۰۴-۱۴۰۳

https://dbsbu.github.io

فصل ۶: مباحث منتخب



تعریف تابع در PostgreSQL

تابع در PostgreSQL، یک قطعه کد قابل استفاده مجدد است که مجموعه‌ای از دستورات را برای انجام یک وظیفهٔ مشخص اجرا می‌کند. توابع می‌توانند ورودی دریافت کنند، یک یا چند عملیات را انجام دهند و خروجی بازگردانند. این توابع ممکن است داخلی (Built-in) باشند، مانند تابع تجمعی SUM() که مجموع مقادیر یک ستون را محاسبه می‌کند، یا توسط کاربر ایجاد شوند که در این صورت به آن‌ها توابع تعریف‌شده توسط کاربر گفته می‌شود.

توابع تعریف‌شده توسط کاربر (User-defined Functions) به توسعه‌دهندگان اجازه می‌دهند تا منطق پیچیده را در قالب ساختارهای ماژولار ذخیره کرده و در سراسر پایگاه داده به‌صورت مکرر از آن استفاده کنند.

دستور CREATE FUNCTION در PostgreSQL

PostgreSQL به شما اجازه می‌دهد تا با استفاده از دستور CREATE FUNCTION یک تابع جدید ایجاد کنید. این تابع، تابعی تعریف‌شده توسط کاربر نامیده می‌شود زیرا توسط شما یا دیگر توسعه‌دهندگان ایجاد شده و به‌صورت پیش‌فرض در PostgreSQL وجود ندارد. تابع تعریف‌شده توسط کاربر یک تابع سفارشی است که مانند یک تابع داخلی، یک کار خاص را انجام می‌دهد. این تابع پارامترهایی به‌عنوان ورودی می‌گیرد، یک یا چند دستور SQL را اجرا می‌کند و یک یا چند مقدار را بازمی‌گرداند.

در ادامه، نحوه پایه‌ای دستور CREATE FUNCTION نمایش داده شده است:

```
CREATE [OR REPLACE] FUNCTION function_name(param1 type, param2 type, ...)
RETURNS return_type AS
$$
BEGIN
-- عملیات
 RETURN result;
END;
 $$ LANGUAGE plpgsql;
```

در این نحو:

CREATE [OR REPLACE] FUNCTION به **PostgreSQL** دستور می‌دهد تا یک تابع جدید ایجاد کند. اگر نام تابع از قبل وجود داشته باشد، گزینه **OR REPLACE** آن را جایگزین می‌کند.

function_name (parameters) نام تابع به‌همراه پارامترها است.

RETURNS return_type نوع داده‌ای را مشخص می‌کند که تابع بازخواهد گرداند.

**AS $$ ... $$** بدنه تابع را نشان می‌دهد که درون رشته‌هایی با علامت دلار ($$) محصور شده است.

مثال ۱. فرض کنید پایگاه داده زیر را داریم:

```
CREATE TABLE students (
  id SERIAL PRIMARY KEY,
  fname TEXT,
  lname TEXT,
  birth_year INT,
  grade NUMERIC
);

INSERT INTO students (fname, lname, birth_year, grade) VALUES
('علی', 'رضایی', ۱۳۸۳, ۱۸/۵),
('نگین', 'حسینی', ۱۳۸۲, ۱۵/۷۵),
('سجاد', 'مرادی', ۱۳۸۰, ۱۳/۰),
('مریم', 'اکبری', ۱۳۸۴, ۱۹/۲۵),
('احمد', 'قاسمی', ۱۳۷۹, ۱۲/۵);
```

تابع محاسبه سن فعلی دانشجو

```
CREATE OR REPLACE FUNCTION get_age(birth_year INT)
RETURNS INT AS $$
BEGIN
  RETURN 1403 - birth_year;
END;
$$ LANGUAGE plpgsql;
```

استفاده در جدول:

```
SELECT
  fname,lname, birth_year, get_age(birth_year) AS age
FROM students;
```

خروجی:

fname	lname	birth_year	age
علی	رضایی	1383	20
نگین	حسینی	1382	21
سجاد	مرادی	1380	23
مریم	اکبری	1384	19
احمد	قاسمی	1379	24

استفاده برای یک داده خاص:

```
select get_age(1371)
```

خروجی:

```
get_age
32
```

مثال ۲. تابعی که دانشجویان با نمره بالاتر از مقدار ورودی را برمی‌گرداند.

```
CREATE OR REPLACE FUNCTION get_students_by_grade(min_grade NUMERIC)
RETURNS SETOF  students AS $$
BEGIN
  RETURN QUERY
  SELECT * FROM students
  WHERE grade >= min_grade;
END;
$$ LANGUAGE plpgsql;
```

RETURNS SETOF students: مشخص می‌کند که خروجی تابع، مجموعه‌ای از سطرهای جدول **students** است (یعنی نه فقط یک رکورد، بلکه ممکن است چندین ردیف از جدول بازگردانده شوند).
RETURN QUERY: این دستور در **PL/pgSQL** برای بازگرداندن نتایج یک کوئری استفاده می‌شود، به‌خصوص زمانی که تابع از نوع **SETOF** است.
SELECT * FROM students WHERE grade >= min_grade: این کوئری تمام دانش‌آموزانی را از جدول **students** انتخاب می‌کند که نمره آن‌ها برابر یا بیشتر از مقدار **min_grade** باشد. این نتیجه به عنوان خروجی تابع بازگردانده می‌شود.

نحوه‌ی استفاده از تابع

به صورت مستقیم

```
SELECT * FROM get_students_by_grade(15.0);
```

خروجی:

id	fname	lname	birth_year	grade
1	علی	رضایی	1383	18.5
2	نگین	حسینی	1382	15.75
4	مریم	اکبری	1384	19.25

همراه با شرط دیگر

```
SELECT *
FROM get_students_by_grade(16.0)
WHERE birth_year > 1381;
```

خروجی:

id	fname	lname	birth_year	grade
1	علی	رضایی	1383	18.5
4	مریم	اکبری	1384	19.25

مثال ۲. دانشجویانی که در سال خاصی به دنیا آمده‌اند

```
CREATE OR REPLACE FUNCTION get_students_by_birthyear(year INT)
RETURNS SETOF students AS $$
BEGIN
  RETURN QUERY
  SELECT * FROM students
  WHERE birth_year = year;
END;
$$ LANGUAGE plpgsql;
```

استفاده:

```
SELECT * FROM get_students_by_birthyear(1383);
```

خروجی:

id	fname	lname	birth_year	grade
1	علی	رضایی	۱۳۸۳	۱۸٫۵

مثال ۳. فقط ۳ دانشجوی با بالاترین معدل

```
CREATE OR REPLACE FUNCTION top_3_students()
RETURNS SETOF students AS $$
BEGIN
  RETURN QUERY
  SELECT * FROM students
  ORDER BY grade DESC
  LIMIT 3;
END;
$$ LANGUAGE plpgsql;
```

استفاده

```
SELECT * FROM top_3_students();
```

خروجی:

id	fname	lname	birth_year	grade
4	مریم	اکبری	1384	19.25
1	علی	رضایی	1383	18.5
2	نگین	حسینی	1382	15.75

ساختار شرطی IF ... THEN ... ELSE

یکی از مهم‌ترین ابزارها برای کنترل جریان اجرای برنامه، ساختار شرطی IF است. این ساختار به شما امکان می‌دهد بسته به برقرار بودن یا نبودن یک شرط مشخص، بخش‌های مختلفی از کد اجرا شوند. در PostgreSQL، ساختار شرطی IF بسیار شبیه زبان‌های برنامه‌نویسی دیگر است.

نحو کلی ساختار IF

```
IF condition THEN
    -- دستورات زمانی که شرط برقرار است
ELSIF another_condition THEN
    -- دستورات زمانی که شرط دوم برقرار است
ELSE
    -- دستورات زمانی که هیچ شرط قبلی برقرار نیست
END IF;
```

- **IF condition THEN** شرطی که بررسی می‌شود. اگر این شرط درست **(TRUE)** باشد، بخش دستورات بعد از آن اجرا می‌شود.
- **ELSIF** (اختیاری) برای بررسی شرط‌های بعدی زمانی که شرط‌های قبلی برقرار نبوده‌اند استفاده می‌شود. می‌توان چندین**ELSIF** پشت سر هم آورد.
- **ELSE** (اختیاری) این بخش زمانی اجرا می‌شود که هیچ‌یک از شرط‌های قبلی برقرار نباشند.
- **END IF;** انتهای ساختار شرطی را مشخص می‌کند.

مثال. نمایش یک پیغام براساس نمره دانشجو

```
CREATE OR REPLACE FUNCTION grade_status(g NUMERIC)
RETURNS TEXT AS $$
BEGIN
  IF g >= 18 THEN
    RETURN 'عالی';
  ELSIF g >= 15 THEN
    RETURN 'خوب';
  ELSIF g >= 12 THEN
    RETURN 'قابل قبول';
  ELSIF g >= 10 THEN
    RETURN 'ضعیف';
  ELSE
    RETURN 'مردود';
  END IF;
END;
$$ LANGUAGE plpgsql;
```

استفاده به صورت مستقیم:

```
SELECT grade_status(13);
```

خروجی:

```
 grade_status

قابل قبول
```

استفاده با SELECT در کوئری:

```
SELECT
  fname, lname, grade, grade_status(grade) AS وضعیت
FROM students;
```

خروجی:

fname	lname	grade	وضعیت
علی	رضایی	18.5	عالی
نگین	حسینی	15.75	خوب
سجاد	مرادی	13.0	قابل قبول
مریم	اکبری	19.25	عالی
احمد	قاسمی	12.5	قابل قبول

ساختار شرطی CASE

ساختار CASE یکی از مهم‌ترین ابزارها در SQL برای پیاده‌سازی منطق شرطی در کوئری‌ها است. این ساختار، عملکردی مشابه IF ELSE در زبان‌های برنامه‌نویسی دارد، با این تفاوت که به‌صورت درون‌خطی (inline) در کوئری‌های SQL مورد استفاده قرار می‌گیرد.

```
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  ...
  ELSE default_result
END
```

- Condition شرطی که بررسی می‌شود.
- Result مقداری که در صورت درست بودن شرط بازگردانده می‌شود.
- ELSE مقدار پیش‌فرض (در صورتی که هیچ‌یک از شرط‌ها برقرار نباشند).
- END پایان عبارت CASE.

مثال. دسته‌بندی نمرات دانشجو

```
SELECT
  fname, lname, grade,
  CASE
    WHEN grade >= 18 THEN 'عالی'
    WHEN grade >= 15 THEN 'خوب'
    WHEN grade >= 12 THEN 'متوسط'
    ELSE 'ضعیف'
  END AS وضعیت
FROM students;
```

خروجی:

fname	lname	grade	وضعیت
علی	رضایی	18.5	عالی
نگین	حسینی	15.75	خوب
سجاد	مرادی	13.0	متوسط
مریم	اکبری	19.25	عالی
احمد	قاسمی	12.5	متوسط

دستور UPSERT

در PostgreSQL، عملیات UPSERT به معنای "درج یا به‌روزرسانی" است. این عملیات زمانی کاربرد دارد که بخواهیم رکوردی را در جدول درج کنیم، اما اگر رکوردی با همان کلید (مثلا کلید اصلی یا یکتا) از قبل وجود داشت، به جای خطا، اطلاعات آن را به‌روزرسانی کنیم.

در SQL استاندارد، چنین عملیاتی نیاز به بررسی‌های دستی دارد، اما PostgreSQL از نسخه ۹/۵ به بعد، به‌صورت داخلی از دستور INSERT ... ON CONFLICT برای UPSERT پشتیبانی می‌کند. نحو پایه‌ی دستور UPSERT به صورت زیر است:

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...)
ON CONFLICT (conflict_target)
DO UPDATE SET column1 = value1, ...
```

- **conflict_target ستونی است که احتمال دارد باعث ایجاد تضاد (conflict) شود، معمولا کلید یکتا یا کلید اصلی.**
- **اگر رکورد جدید باعث ایجاد conflict شود، بجای آن DO UPDATE انجام می‌شود.**
- **اگر هیچ conflict ایجاد نشود، داده به‌صورت عادی درج می‌شود.**

مثال ۱: اگر رکورد با id = 3 وجود داشت، نمره‌اش را به مقدار جدید ۱۶ تغییر بده

```
INSERT INTO students (id, fname, lname, birth_year, grade)
VALUES(۳, 'سجاد', 'مرادی', ۱۳۸۰, ۱۶/۰)
ON CONFLICT (id)
DO UPDATE SET grade = 16.0;
```

- **اگر دانش‌آموزی با شناسه‌ی ۳ وجود نداشته باشد، رکورد جدید اضافه می‌شود.**
- **اگر وجود داشته باشد، فقط نمره‌ی او به ۱۶/۰ تغییر پیدا می‌کند.**

مثال ۲: اگر شناسه تکراری بود، هیچ کاری انجام نده

```
INSERT INTO students (id, fname, lname, birth_year, grade)
VALUES(۳, 'سجاد', 'مرادی', ۱۳۸۰, ۱۶/۰)
ON CONFLICT (id)
DO NOTHING;
```

- اگر id = 3 وجود داشته باشد، رکورد نادیده گرفته می‌شود.
- اگر وجود نداشته باشد، رکورد درج می‌شود.

کار با JSON در PostgreSQL

JSON چیست؟

JSON یا JavaScript Object Notation، یک قالب متنی سبک‌وزن و قابل‌خواندن برای انسان و ماشین است که برای تبادل داده بین کلاینت و سرور، اپلیکیشن‌ها و سرویس‌ها به‌صورت گسترده استفاده می‌شود.

مثال:

```
{
  "name": "میلاد",
  "age": ۳۲,
  "skills": ["Python", "SQL", "ML"],
  "address": {
    "city": "تهران",
    "zipcode": "12345"
  }
}
```

چرا JSON مهم است؟

- **زبان-بی‌طرف است:** از آنجا که ساختار JSON مستقل از زبان برنامه‌نویسی است، با اکثر زبان‌ها مانند Python، JavaScript، PHP، Java و ... به‌خوبی کار می‌کند.
- **ساختارمند ولی انعطاف‌پذیر است:** برخلاف جداول سنتی که ساختار ثابتی دارند، JSON اجازه می‌دهد که ساختار داده‌ها پویا و تو در تو باشد.
- **استاندارد تبادل داده در وب است:** اکثر APIها به‌ویژه RESTful از JSON برای ارسال و دریافت داده استفاده می‌کنند.
- **خوانا برای انسان و ماشین:** هم توسعه‌دهنده می‌تواند آن را بفهمد، هم سیستم‌ها می‌توانند به‌راحتی آن را پردازش کنند.

چرا ذخیره JSON در پایگاه‌داده اهمیت دارد؟

در گذشته، بانک‌های اطلاعاتی رابطه‌ای مانند MySQL و PostgreSQL کلاسیک فقط با داده‌های ساخت‌یافته (جدولی) کار می‌کردند. اما امروزه نیازها تغییر کرده:

- **داده‌های دریافتی از APIها معمولاً به شکل JSON هستند.**
- **برخی داده‌ها ساختار ثابتی ندارند یا به‌صورت پویا تولید می‌شوند.**

پس نیاز است که پایگاه داده‌ای بتواند داده‌های نیمه‌ساخت‌یافته یا بدون ساختار را ذخیره، جست‌وجو، و پردازش کند.

PostgreSQL از دو نوع داده‌ی مربوط به JSON پشتیبانی می‌کند:

نوع داده	توضیح
json	داده‌های JSON به شکل متنی ذخیره می‌شوند (فرمت اصلی حفظ می‌شود).
jsonb	داده‌ها به صورت باینری ذخیره شده و برای جست‌وجو و فیلتر بهینه‌سازی شده‌اند.

معمولا jsonb پیشنهاد می‌شود چون سریع‌تر است.

در PostgreSQL، برای دسترسی به داده‌های درون فیلدهای JSON یا JSONB از دو عملگر زیر استفاده می‌کنیم:

عملگر	خروجی	نوع داده	کاربرد
->	JSON	jsonb یا json	مقدار به‌صورت آبجکت JSON برمی‌گرداند
->>	متن	text	مقدار به‌صورت رشته متنی برمی‌گرداند

->

نحو:

```
json_column -> 'key'
```

برای استخراج زیرآبجکت‌ها یا آرایه‌ها یا مقادیر JSON از یک کلید استفاده می‌شود.

->>

نحو:

```
json_column ->> 'key'
```

برای استخراج مقدار نهایی به‌صورت رشته متنی استفاده می‌شود، مخصوصاً زمانی که می‌خواهی آن مقدار را نمایش بدهی، فیلتر کنی یا با متن مقایسه کنی.

?

بررسی وجود کلید در یک شی JSONB

نحو با مثال:

```
info ? 'city'
```

فقط برای jsonb تعریف شده. بررسی می‌کند که آیا کلیدی به نام city وجود دارد یا نه.

عملگر تبدیل نوع ::

مثال تبدیل مقدار متنی به عدد صحیح.

```
(info->>'age')::INT
```

از ستون JSON، کلید "age" را به صورت رشته می‌گیرد و سپس آن را به عدد صحیح (INT) تبدیل می‌کند.

این ترکیب زمانی استفاده می‌شود که می‌خواهی با مقادیر عددی در JSON عملیات ریاضی یا مقایسه انجام دهی.

jsonb_array_elements(jsonb)

استخراج آیتم‌های یک آرایه JSON به‌صورت سطر به سطر

وقتی یک ستون شامل آرایه‌ای از داده‌ها (در قالب JSONB) است، با این تابع می‌توان آن آرایه را به‌صورت لیستی از ردیف‌ها گسترش داد تا روی هر عنصر جداگانه کار کرد (مثل فیلتر، جستجو، گروه‌بندی و ...).

jsonb_array_elements_text(jsonb)

همان عملکرد ولی خروجی از نوع متنی (text)

ایجاد جدول با فیلد JSON

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  name TEXT,
  info JSONB
);
```

درج داده در فیلد JSON

```
INSERT INTO users (name, info) VALUES
('علی', '{"age": 25, "city": "تهران"}'),
('نگار', '{"age": 30, "city": "اصفهان", "skills": ["SQL", "Python"]}'),
('سینا', '{"age": 28, "city": "تبریز", "married": false}'),
('مریم', '{"age": 35, "city": "شیراز", "skills": ["Java", "HTML", "CSS"]}'),
('حسین', '{"age": 40, "city": "مشهد", "job": "مدیر پروژه"}'),
('رعنا', '{"age": 22, "city": "کرج", "education": {"degree": "لیسانس", "field": "علوم کامپیوتر"}}'),
('پویا', '{"age": 33, "city": "رشت", "married": true, "children": 2}'),
('سحر', '{"age": 29, "city": "یزد", "languages": ["انگلیسی", "فرانسوی"]}'),
('امیر', '{"age": 27, "city": "اراک", "skills": ["C#", "SQL"], "job": "توسعه‌دهنده"}'),
('الهه', '{"age": 31, "city": "همدان", "address": {"street": "بلوار آزادی", "zipcode": "654321"}}');
```

کوئری گرفتن از داده‌های JSON

۱. نمایش نام و شهر هر کاربر

```
SELECT
  name,
  info->>'city' AS city
FROM users;
```

خروجی:

name	city
علی	تهران
نگار	اصفهان
سینا	تبریز
مریم	شیراز
حسین	مشهد
رعنا	کرج
پویا	رشت
سحر	یزد
امیر	اراک
الهه	همدان

۲. نمایش کاربران بالای ۳۰ سال

```
SELECT
  name,
  info->>'age' AS age
FROM users
WHERE (info->>'age')::INT > 30;
```

خروجی:

name	age
مریم	۳۵
حسین	۴۰
پویا	۳۳
الهه	۳۱

۳. کاربرانی که مهارت Python دارند

```
SELECT
  name,
  info->'skills' AS skills
FROM users
WHERE info->'skills' ? 'Python';
```

خروجی:

name	skills
نگار	['SQL', 'Python']

۴. کاربرانی که وضعیت تأهل آن‌ها مشخص شده

```
SELECT
  name,
  info->>'married' AS married
FROM users
WHERE info ? 'married';
```

خروجی:

name	married
سینا	false
پویا	true

۵. استخراج رشته تحصیلی کاربران

```
SELECT
  name,
  info->'education'->>'field' AS field_of_study
FROM users
WHERE info->'education' ? 'field';
```

خروجی:

name	field_of_study
رعنا	علوم کامپیوتر

۶. کاربرانی که دارای فرزند هستند

```
SELECT
  name,
  info->>'children' AS children
FROM users
WHERE info ? 'children';
```

خروجی:

۷. مهارت‌های کاربران را به صورت سطر به سطر نمایش بده

```
SELECT
  name,
  jsonb_array_elements_text(info->'skills') AS skill
FROM users
WHERE info ? 'skills';
```

خروجی:

name	skill
نگار	SQL
نگار	Python
مریم	Java
مریم	HTML
مریم	CSS
امیر	C#
امیر	SQL

۸. شمارش تعداد کاربرانی که شغل دارند

```
SELECT COUNT(*) AS count_with_job
FROM users
WHERE info ? 'job';
```

خروجی

count_with_job
2

۹. شمارش تعداد مهارت‌های هر کاربر

```
SELECT
  name,
  COUNT(*) AS skill_count
FROM users,
  jsonb_array_elements_text(info->'skills') AS skill
GROUP BY name;
```

خروجی:

name	skill_count
نگار	۲
امیر	۲
مریم	۳

۱۰. فیلتر کاربران دارای مهارت خاص (مثلاً SQL) با استفاده از ساب‌کوئری

```
SELECT name
FROM users
WHERE EXISTS (
  SELECT 1
  FROM jsonb_array_elements_text(info->'skills') AS skill
  WHERE skill = 'SQL'
);
```

خروجی:

```
name
نگار
امیر
```

۱۱. کاربران دارای مهارت Python و SQL همزمان

```
SELECT name
FROM users
WHERE EXISTS (
  SELECT 1 FROM jsonb_array_elements_text(info->'skills') AS s WHERE s = 'Python'
)
AND EXISTS (
  SELECT 1 FROM jsonb_array_elements_text(info->'skills') AS s WHERE s = 'SQL'
);
```

خروجی:

```
name
نگار
```